BIAX Corporation v. Intel
Civil Action No. 2:05-cv-184-TJW

EXHIBIT 2
(PART 4)
FIRST AMENDED COMPLAINT FOR PATENT INFRINGEMENT

tional cache high speed RAM as found in conventional superminicomputers.

The tag generation circuit **1612** provides a unique identification code (ID) for attachment to each instruction before storage of that instruction in the designated RAM **1610**. The assigning of process identification tags to instructions stored in cache circuits is conventional and is done to prevent aliasing of the instructions. "Cache Memories" by Alan J. Smith, ACM Computing Surveys, Vol. 14, September, 1982. The tag comprises a sufficient amount of information to uniquely identify it from each other instruction and user. The instructions already include the IFT and LPN, so that subsequently, when instructions are retrieved for execution, they can be fetched based on their firing times. As shown in Table 16, below, each instruction containing the extended information and the hardware tag is stored as shown for the above example:

TABLE 16

CACHE0:	I4(T16)(PE2)(ID2)
CACHE1:	I2(T17)(PE0)(ID3)
CACHE2:	I0(T16)(PE0)(ID0)
	I5(T17)(PE1)(ID4)
CACHE3:	I1(T16)(PE1)(ID1)
	I3(T18)(PE0)(ID5)

As stated previously, the purpose of the cache partition circuits **1552** is to provide a high speed buffer of the between the slow main memory **610** and the fast processor elements **640**. Typically, the cache RAM **1610** is a high speed memory capable of being quickly accessed. If the RAM **1610** were a true associative memory; as can be witnessed in Table 16, each RAM **1610** could be addressed based upon instruction firing times (IFTs). At the present, such associative memories are not economically justifiable and an IFT to cache address translation circuit **1620** must be utilized. Such a circuit is conventional in design and controls the addressing of - each RAM **1610** over bus **1520*d***. The purpose of circuit **1620** is to generate the RAM address of the desired instructions given the instruction firing time. Hence, for instruction firing time **T16**, CACHE0, CACHE2, and CACHE3, as seen in Table 16, would produce instructions **I4**, **I0**, and **I1** respectively. Hence, when the cache RAMs **1610** are addressed, those instructions associated with a specific firing time are delivered into a tag compare and privilege check circuit **1630**.

The purpose of the tag compare and privilege check circuit **1630** is to compare the hardware tags (ID) to the generated tags to verify that the proper instruction has been delivered. Again, the tag is generated through a generation circuit **1632** which is interconnected to the tag compare and privilege check circuit **1630** over line **1520*e***. A privilege check is also performed on the instruction delivered to verify that the operation requested by the instruction is permitted given the privilege status of the process (e.g., system program, application program, etc.) This is a conventional check performed by computer processors which support multiple levels of processing states. The hit/miss circuit **1640** determines which RAMs **1610** have delivered the proper instructions to the PIQ bus interface unit **1544** in response to a specific instruction fetch request.

For example, and with reference back to Table 16, if the RAMs **1610** are addressed by circuit **1620** for instruction firing time **T16**, CACHE0, CACHE2, and CACHE3 would respond with instructions thereby

comprising a hit indication on those cache partitions. Cache **1** would not respond and that would constitute a miss indication and this would be determined by circuit **1640** over line **1520*g***. Likewise, for instruction firing time **T16** three instructions are delivered. Each addressed instruction is then delivered over bus **1632** into the SCSM attacher **1650** wherein any dynamic SCSM information is added onto each instruction by the hardware **1650**.

When all of the instructions associated with an individual firing time have been read from the RAM **1610**, the hit and miss circuit **1640** over lines **1646** informs the instruction cache control unit **1660** of this information. The instruction cache control unit **1660** contains the next instruction firing time register **1518** which increments the instruction firing time to the next value. Hence, in the example, upon the completion of reading all instructions associated with instruction firing time **T16**, the instruction cache control unit **1660** increments to the next firing time, **T17** and delivers this information over lines **1664** to the access resolution circuit **1670**, and over lines **1666** to the tag compare and privilege check circuit **1630**. Also note that there may be firing times which have no valid instructions, possibly due to operational dependencies detected by TOLL. In this case, no instructions would be fetched from the cache and transmitted to the PIQ.

The present invention can be a multiuser computer architecture capable of supporting several users simultaneously in both time and space. In previous prior art approaches (CDC, IBM, etc.), multiuser support was accomplished by timesharing the processor(s). In other words, the processors were shared in time. In this system, multiuser support is accomplished by assigning an LRD to each user that is given time on the processor elements. Thus, there is a spatial aspect to the sharing of the processor elements. The operating system of the machine would assign users to the LRDs in a timeshared manner, thereby adding the temporal dimension to the sharing of the processors.

Hence, multiuser support is accomplished by the multiple LRDs, the use of context free processor elements, and the multiple context support present in the register and condition code files. As several users may be executing in the processor elements at a time, additional pieces of information must be attached to each instruction prior to its execution in order to uniquely identify the instruction source and any resources that it may use. For example, a register identifier must contain the procedural level and context identifier as well as the actual register number. Memory addresses must also contain the LRD identifier that the instruction was issued from in order to get routed through the data cache interconnection network to the appropriate data cache.

The information comprises two components - static and dynamic and, as maintained, is termed shared context storage mapping (SCSM). The static information is composed of information that the compiler or TOLL can glean from the instruction stream. For example, the register window tag would be generated statically and attached to the instruction prior to its being received by an LRD.

The dynamic information is hardware attached to the instruction by the LRD prior to its issuance to the processors. This information is composed of the context/LRD identifier that is issuing the instruction, the current procedural level of the instruction, the process identifier of the current instruction stream, and the instruction status information that would normally be contained in the processors of a system with processors that are not context free. This later information would be composed of error masks, floating point format modes, rounding modes and so on.

The operation of the circuitry in FIG. **16** can be summarized as follows. One or more execution sets are delivered into the instruction cache circuitry of FIG. **16**, the header information for each set is delivered to one or more successive cache partitions and is routed into the control unit **1660**. The remaining instructions in the execution set are then individually, on a round robin basis, routed into each successive cache partition unit **1552**, a hardware identification tag is attached to each instruction and it is stored in RAM **1610**. As previously discussed, each execution set is of sufficient length to minimize instruction cache defaults and the RAM **1610** is of sufficient size to store the execution sets. When the processor elements require the information, the instructions stored in the RAMs **1610** are read out, the identification tags are verified and the privilege status checked. The number and cache locations of valid instructions matching the appropriate IFTs are determined. The instructions are then delivered to PIQ bus interface unit **1544**. The information that is delivered to the PIQ bus interface unit **1544** is set forth in Table 17 including the identification tag (ID) and the hardware added SCSM information.

TABLE 17

CACHE0:	I4(T16)(PE2)(ID2)(SCSM0)
CACHE1:	I2(T17)(PE0)(ID3)(SCSM1)
CACHE2:	I0(T16)(PE0)(ID0)(SCSM2)
	I5(T17)(PE1)(ID4)(SCSM3)
CACHE3:	I1(T16)(PE1)(ID1)(SCSM4)
	I3(T18)(PE0)(ID5)(SCSM5)

In FIG. **17**, the details of the PIQ bus interface unit **1544** and the PIQ buffer unit **1560** are set forth. These circuits function as follows. The PIQ bus interface unit **1544** receives instructions as set forth in Table **17**, above, over leads **1536**. These instructions access, in parallel, a series of bus interface units (BIUs) **1700**. The bus interface units **1700** are interconnected together in a full access non-blocking network by means of connections **1710** and **1720** over lines **1552** to the PIQ buffer unit **1560**. Each bus interface unit (BIU) **1700** is a conventional address comparison circuit composed of: TI 74L85 4 bit magnitude comparators, Texas Instruments Company, P.0. Box 225012, Dallas, Texas 75265. In the matrix multiply example, for instruction firing time T16, CACHE0 contains instruction I4 and CACHE3 (corresponding to CACHE N in FIG. 17) contains instruction I1. The logical processor number assigned to instruction I4 is PE2 and, therefore, the logical processor number PE2 activates a select (SEL) signal of the bus interface unit **1700** for processor instruction queue 2 (BIU3). In this example, only BIU3 is activated and the remaining bus interface units **1700** are not activated. Likewise, for CACHE3 (CACHE N), BIU2 is activated for processor instruction QUEUE **1**.

The PIQ buffer unit **1560** is comprised of a number of processor instruction queues **1730** which store the instructions received from the PIQ bus interface unit **1544** in a first in-first out (FIFO) fashion as shown in Table 18:

TABLE 18

PIQ0	PIQ1	PIQ2	PIQ3
I0	I1	I4	—
I2	—	—	—
I3	—	—	—

In addition to performing instruction queueing functions, the PIQs **1730** also keep track of the execution status of each instruction that are issued to the processor elements **640**. In an ideal system, instructions could be issued to the processor elements every clock cycle without worrying about whether or not the instructions have finished execution. However, the processor elements **640** in the system may not be able to complete an instruction every clock cycle due to exceptional conditions occurring, such as a data cache miss and so on. As a result, each PIQ **1730** tracks all instructions that it has issued to the processor elements **640** that are still in execution. The primary result of this tracking is that the PIQ's **1730** perform the instruction clocking function for the LRD **620**. In other words, the PIQs **1730** determine when the next firing time register can be updated when executing straightline code. This in turn begins a new instruction fetch cycle.

Instruction clocking is accomplished by having each PIQ **1730** form an instruction done signal that specifies that the instruction(s) issued by a given PIQ have executed or proceeded to the next stage in the case of pipelined PEs. This is then combined with all other PIQs instruction done signals from this LRD and used to gate the increment signal that increments the next firing time register. These signals are delivered over lines **1564** to the instruction cache control **1518**.

The details of the PIQ processor assignment circuit **1570** is set forth in FIG. 18. The PIQ processor assignment circuit **1570** contains a set of network interface units (NIUs) **1800** interconnected in a full access switch to the PE-LRD network **650** and then to the various processor elements **640**. Each network interface unit (NIU) **1800** is comprised of the same circuitry as the bus interface units (BIU) **1700** of FIG. **17**. In normal operation, the processor instruction queue (PIQ0) directly accesses processor element **0** and NIU0 is activated and the remaining network interface units NIU1, NIU2, NIU3, for PIQ are deactivated. Likewise, processor instruction PIQ3 normally accesses processor element **3** having its NIU3 activated and the corresponding NIU0, NIU1, deactivated. The activation of which network interface unit **1800** is under the control of an instruction select and assignment unit **1810**.

This unit **1810**, receives signals from the PIQs within the LRD over lines **1811** that the unit **1810** is a member of, and from all other LRDs unit **1810** over lines **1813**, and from the processor elements **640** through the network **650**. Each PIQ furnishes the unit a signal that corresponds to "I have an instruction that is ready to be assigned to a processor." The other units furnish this unit and every other unit a signal that corresponds to "My PIQ #x has an instruction ready to be assigned to a processor." Finally, the processor elements furnish the unit and all other units in the system a signal that corresponds to "I can accept a new instruction."

The unit **1810** transmits signals to the PIQs of the LRD over lines **1811**, the network interface units **1800** of the LRD and the other units **1810** of the other LRDs in the system over lines **1813**. The unit transmits a signal to each PIQ that corresponds to "Gate your instruction onto the PE-PIQ interface bus (1562)." The unit transmits a select signal to the network interface units **1800**. Finally, the unit transmits a signal that corresponds to "I have used processor element #x" for each processor in the system to each other unit **1810** in the system.

In addition, each unit **1810** in each LRD has associated with it a priority that corresponds to the priority of the LRD. This is used to order the LRDs into an ascending order from zero to the number of LRDs in the system. The method used for assigning the processor elements is as follows. Given that the LRDs are ordered, many allocation schemes are possible (e.g., round robin, first come first served, time slice, etc.). However, these are implementation details and do not impact the functionality of this unit under the teachings of the present invention.

Consider the LRD with the current highest priority. This LRD gets any and all processor elements that it requires and assigns the instructions that are ready to be executed to the available processor elements in any manner whatsoever due to the fact that the processor elements are context free. Typically, however, assuming that all processors are functioning correctly, instructions from PIQ #0 are routed to processor element #0, provided of course, processor element #0 is available.

The unit **1810** in the highest priority LRD then transmits this information to all other **1810** units in the system. Any processors left open are then utilized by the next highest priority LRD with instructions that can be executed. This allocation continues until all processors have been assigned. Hence, processors may be assigned on a priority basis in a daisy chained manner.

If a particular processor element, for example, element **1** has failed, the instruction selective assignment unit **1810** can deactivate that processor element by deactivating all network instruction units corresponding to NIU1. It can then, through hardware, reorder the processor elements so that, for example, processor element **2** receives all instructions logically assigned to processor element **1**, processor element **3** is now assigned to receive all instructions logically assigned to processor **2**. Indeed, redundant processor elements and network interface units can be provided to the system to provide for a high degree of fault tolerance.

Clearly, this is but one possible implementation. Other methods are also realizable.

b. Branch Execution Unit (BEU)

The details of a Branch Execution Unit (BEU) **1548** are shown in FIG. **19**. The Branch Execution Unit (BEU) **1548** is the unit in the present invention responsible for the execution of all branch instructions which occur at the end of each basic block. There is one BEU **1548** per context support hardware in the LRD and so, with reference back to FIG. **6** "n" contexts would require "n" BEUs. The reasoning being that each BEU **1548** is of simple design and, therefore, the cost of sharing it between contexts would be more expensive than allowing each context to have its own BEU.

Under the teachings of the present invention it is desired that branches be executed as fast as possible. In order to accomplish this, the instructions do not perform conventional next instruction address computation and with the exception of the subroutine return branches, already contain the full target or next branch address. In other words, the target address is static when the branch is executed, there is no dynamic gener-

ation of branch target addresses. Further, the target address is fully contained within the branch, i.e. all branch addresses are known at program preparation time in the TOLL output and, as a result, are directed to absolute addresses only. When a target address has been selected to be taken as a result of a branch other than the aforementioned subroutine return branch, the address is read out of the instruction and placed directly into the next instruction fetch register.

Return from subroutine branches are handled in a slightly different fashion. In order to understand the subroutine return branch, discussion of the subroutine call branch is required. A subroutine call is an unconditional branch whose target address is determined at program preparation time, as described above. When the branch is executed, a return address is created and stored. The return address is normally the address of the instruction following the subroutine call. The return address can be stored in a stack in memory or in other storage local to the branch execution unit. In addition, the execution of the subroutine call increments the procedural level counter.

The return from subroutine branch is also an unconditional branch. However, rather than containing the target address within the instruction, this type of branch reads the previously stored return address from the storage, decrements the procedural level counter, and loads instruction fetch register with the return address. The remainder of the disclosure discusses the evaluation and execution of conditional branches. It should be noted that techniques described also apply to unconditional branches, since these are, in effect, conditional branches in which the condition is always satisfied. Further, these same techniques also apply to the subroutine call and return branches, which perform the additional functions described above.

To speed up conditional branches, the determination of whether a conditional branch is taken or not, depends solely on the analysis of the appropriate set of condition codes. Under the teachings of the present invention, there is no evaluation of data performed other than to manipulate the condition codes appropriately. In addition, an instruction generating a condition code that a branch will use can transmit the code to BEU **1548** as well as to the condition code storage. This eliminates the conventional extra time required to wait for the code to become valid in the condition code storage prior to the BEU being able to fetch it.

Also, the present invention makes extensive use of delayed branching. In order to guarantee program correctness, when a branch has executed and its effects are propagated in the system, all instructions that are within the procedural domain of the given branch must have been executed or be in the process of being executed as discussed with the example of Table 6. In other words, the changing of the next instruction pointer (in response to the branch) must take place after the current firing time has been updated to point to the firing time that would have followed the last (temporally executed) instruction governed by this branch. Hence, in the example of Table 6 instruction **I5** at firing time **T17** is delayed until the completion of **T18** which is the last firing time for this basic block. The instruction time for the next basic block is then **T19**.

The functionality of the BEU **1548** can be described as a fourstate state machine:

Stage 1:	Instruction decode
	- Operation decode
	- Delay field decode
	- Condition code access decode
Stage 2:	Condition code fetch/receive
Stage 3:	Branch operation evaluation
Stage 4:	Next instruction fetch location and firing time update

Along with determining the operation to be performed, the first stage also determines how long fetching can continue to take place after receipt of the branch by the BEU, and how the BEU is to access the condition codes for a conditional branch, i.e. are they received or fetched.

The branch instruction is delivered over bus **1546** from the PIQ bus interface unit **1544** into the instruction register **1900** of the BEU **1548**. In FIG. **19** the fields of the instruction register **1900** are designated as: FETCH-/ENABLE, CONDITION CODE ADDRESS, 0P CODE, DELAY FIELD, and TARGET ADDRESS. The instruction register **1900** is connected over lines **1910***a* and **1910***b* to a condition code access unit **1920**, to an evaluation unit **1930** over lines **1910***c*, a delay unit **1940** over lines **1910***d*, and to a next instruction interface **1950** over lines **1910***e*.

Once an instruction has been issued to BEU **1548** from the PIQ bus interface **1544**, instruction fetching must be held up until the value in the delay field has been determined. This value is measured relative to the receipt of the branch by the BEU, i.e. stage **1**. If there are no instructions that may be overlapped with this branch, this field value is zero. In this case, instruction fetching is held up until the outcome of the branch has been determined. If this field is non-zero, instruction fetching may continue for a number of firing times given by the value in this field.

The condition code access unit **1920** is connected to the register file - PE network **670** over lines **1550** and to the evaluation unit **1930** over lines **1922**. The condition code access decode unit **1920** determines whether or not the condition codes must be fetched by the instruction, or whether or not the instruction that determines the branch condition delivers them. As there is only one instruction per basic block that will determine the conditional branch, there will never be more than one condition code received by the BEU. As a result, the actual timing of when the condition code is received is not important. If it comes earlier than the branch, no other codes will be received prior to the execution of the branch. If it comes later, the branch will be waiting and the codes received will always be the right ones.

The evaluation unit **1930** is connected to the next instruction interface **1950** over lines **1932**. The next instruction interface **1950** is connected to the context control circuit **1518** over lines **1549***b* and to the delay unit **1940** over lines **1942**. The evaluation stage combines the condition codes according to a Boolean function that represents the condition. The final stage either enables the fetching of the stream to continue if a conditional branch is not taken, or, loads up the next instruction pointer if the branch is taken. Finally the delay unit **1940** is also connected to the instruction cache control unit **1518** over lines **1549**.

The impact of a branch in the instruction stream can be described as follows. Instructions, as discussed, are set to their respective PIQ's **1730** by analysis of the

resident logical processor number (LPN). Instruction fetching can be continued until a branch is seen, i.e. an instruction is delivered into the instruction register **1900** of the BEU **1548**. At this point in a conventional system without delayed branching, fetching would be stopped until the resolution of the branch. See, for example, "Branch Prediction Strategies and Branch Target Buffer Design", J.F.K. Lee & A.J. Smith, IEEE Computer Magazine, January, 1984.

In the present system having delayed branching, instructions must continue to be fetched until the point where the next instruction fetch is the last instruction to be fired in the basis block. The time that the branch is executed is the last time that fetching takes place without the possible modification of the next instruction address. Thus, this difference in firing times between when the branch is executed and when the effects of the branch are actually felt corresponds to the number of additional firing times that fetching may be continued.

The impact of the above on the instruction cache is that the BEU **1548** must have access to the next instruction firing time register of the cache controller. The BEU **1548** also controls the initiation or disabling of the fetch process of the instruction cache control **1518** via the instruction cache control unit **1518**. These tasks are accomplished over bus **1549**.

In operation the branch execution unit (BEU) **1548** functions as follows. The branch instruction such as instruction I5 in the example is loaded into the instruction register **1900** from the PIQ bus interface unit **1544**. The instruction register contents then control the operation of BEU **1548**. The FETCH-ENABLE field indicates whether or not the condition code access unit **1920** should retrieve the condition code located at the address stored in the CC-ADX field (i.e. FETCH) or whether the condition code will be delivered by the generating instruction.

If a FETCH is requested, the unit **1920** accesses the register file-PE network **670** (see FIG. **6**) to access the condition code registers **2000** which are shown in FIG. **20**. In FIG. **20**, the condition code registers **2000** for each context are shown in the generalized case. A set of registers CCm are provided for storing condition codes for procedural levels, L. Hence, the condition code registers **2000** are accessed and addressed by the unit **1920** to retrieve pursuant to a FETCH request, the necessary condition code. An indication that the condition code is received by the unit **1920** is delivered over lines **1922** to the evaluation unit **1930** as well as the actual condition code. The OPCODE field delivered to the evaluation unit **1930** in conjunction with the received condition code functions to deliver a branch taken signal over line **1932** to the next instruction interface **1950**. The evaluation unit **1930** is comprised of standard gate arrays such as those from LSI Logic Corporation, 1551 McCarthy Blvd., Milpitas, California 95035.

The evaluation unit **1930** accepts the condition code set that determines whether or not the conditional branch is taken, and under control of the OPCODE field combines the set in a Boolean function to generate the conditional branch taken signal.

The next instruction interface **1950** receives the branch target address from the TARGET-ADX field of the instruction register **1900**. However, the interface **1950** cannot operate until an enable signal is received from the delay unit **1940** over lines **1942**.

The delay unit **1940** determines the amount of time that instruction fetching can be continued after the receipt of a branch by the BEU. Previously, it has been described that when a branch is received by the BEU, instruction fetching continues for one more cycle and then stops. The instructions fetched during this cycle are held up from entering the PIQ **1544** until the length of the delay field has been determined. For example, if the delay field is zero (implying that the branch is to be executed immediately), these instructions must be withheld from the PIQ until it is determined whether or not these are the right instructions to be fetched. Otherwise, (the delay field is nonzero), the instructions would be gated into the PIQ as soon as the delay value was determined to be non-zero. The length of the delay is obtained from DELAY field of the instruction register **1900** and receives clock impulses from the context control **1518** over lines **1549***a*. The delay unit **1940** decrements the value of the delay with the clock pulses and when fully decremented the interface unit **1950** becomes enabled.

Hence, in the discussion of Table 6, instruction **I5** is assigned to firing time **T17** but is delayed until firing time **T18**. During the delay time, the interface **1950** signals the instruction cache control **1518** over line **1549***b* to continue to fetch instructions to finish the current basic block. When enabled, the interface unit **1950** delivers the next address (i.e. the branch execution) for the next basic block into the instruction cache control **1518** over lines **1549***b*.

In summary and for the example on Table 6, the branch instruction **I5** is loaded into the instruction register **1900** during time **T17**. However, a delay of one firing time (DELAY) is also loaded into the instruction register **1900** as the branch instruction cannot be executed until the last instruction **I3** is processed during time **T18**. Hence, when the instruction **I5** is loaded, the branch contained in the TARGET ADDRESS to the next basic block does not take place until the completion of time **T18**. In the meantime, the next instruction interface **1950** issues instructions to the context control **1518** to continue processing the stream of instructions in the basic block. Upon the expiration of the delay, the interface **1950** is enabled, and the branch is executed by delivering the address of the next basic block to the context control **1518**.

c. Processor Elements (PE)

So far in the discussions pertaining to the matrix multiply example, a single cycle processor element has been assumed. In other words, an instruction is issued to the processor element and the processor element completely executes the instruction before proceeding to the next instruction. However, greater performance can be obtained by pipelined processor elements, the tasks performed by TOLL change slightly. In particular, the assignment of the processor elements is more complex than is shown in the previous example. In addition, the hazards that characterize a pipeline must be handled by the TOLL software. The hazards that are present in any pipeline manifest themselves as a more sophisticated set of data dependencies. This can be encoded into the TOLL software by someone skilled in the art. See for example, T.K.R. Gross, Stanford University, 1983, "Code Optimization of Pipeline Constraints", Doctorate Dissertation Thesis.

The assignment of the processors is dependent on the implementation of the pipelines and again, can be performed by someone skilled in the art. The key parameter is determining how data is exchanged between the pipelines. For example, assume that each pipeline contains feedback paths between its stages. In addition, assume that the pipelines can exchange results only through the register sets **660**. Instructions would be assigned to the pipelines by determining sets of dependent instructions that are contained in the instruction stream and then assigning each specific set to a specific pipeline. This minimizes the amount of communication that must take place between the pipelines (via the register set), and hence speeds up the execution time of the program. The use of the logical processor number guarantees that the instructions will execute on the same pipeline.

Alternatively, if there are paths available to exchange data between the pipelines, dependent instructions may be distributed across several pipes instead of being assigned to a single pipe. Again, the use of multiple pipelines and the interconnection network between them that allows the sharing of intermediate results manifests itself as a more sophisticated set of data dependencies imposed on the instruction stream. Clearly, the extension of the teachings of this invention to a pipelined system is within the skill of the art.

In FIG. **21**, the details of the processor elements **640** are set forth for a four-stage pipeline processor element. All processor elements **640** are identical. It is to be expressly understood, that any prior art type of processor element such as a micro-processor or other pipeline architecture could not be used under the teachings of the present invention, because such processors retain the state information of the program they are processing. However, such a processor could be programmed with software to emulate or simulate the type of processor necessary for the present invention. As previously mentioned, each processor element **640** is context-free which differentiates it from conventional processor elements that requires context information. The design of the processor element is determined by the instruction set architecture generated by TOLL and, therefore, is the most implementation dependent portion of this invention from a conceptual viewpoint. In the preferred embodiment shown in FIG. **21**, each processor element pipeline operates autonomously of the other processor elements in the system. Each processor element is homogeneous and is capable of executing all computational and data memory accessing instructions. In making computational executions, transfers are from register to register and for memory interface instructions, the transfers are from memory to registers or from registers to memory.

In FIG. **21**, the four-stage pipeline for the processor element **640** of the present invention includes four discrete instruction registers **2100**, **2110**, **2120**, and **2130**. It also includes four stages: stage **1**, **2140**; stage **2**, **2150**; stage **3**, **2160**, and stage **4**, **2170**. The first instruction register **2100** is connected through the network **650** to the PIQ processor assignment circuit **1570** and receives that information over bus **2102**. The instruction register then controls the operation of stage **1** which includes the hardware functions of instruction decode and register **0** fetch and register **1** fetch. The first stage **2140** is interconnected to the instruction register over lines **2104** and to the second instruction register **2110** over lines **2142**. The first stage **2140** is also connected over bus **2144** to the second stage **2150**. Register **0** fetch and register **1** fetch of stage **1** are connected over lines **2146** and 2148, respectively, to network 670 for access to the register file 660.

The second instruction register 2110 is further interconnected to the third instruction register 2120 over lines 2112 and to the second stage 2150 over lines 2114. The second stage 2150 is also connected over bus 2152 to the third stage 2160 and further has the memory write (MEM WRITE) register fetch hardware interconnected over lines 2154 to network 670 for access to the register file 660 and its condition (CC) code hardware connected over lines 2156 through network 670 to the condition code file 660.

The third instruction register 2120 is interconnected over lines 2122 to the fourth instruction register 2130 and is also connected over lines 2124 to the third stage 2160. The third stage 2160 is connected over bus 2162 to the fourth stage 2170 and is further interconnected over lines 2164 through network 650 to the data cache interconnection network 1590.

Finally, the fourth instruction register 2130 is interconnected over lines 2132 to the fourth stage, and the fourth stage has its store hardware (STORE) output connected over 2172 and its effective address update (EFF. ADD.) hardware circuit connected over 2174 to network 670 for access to the register file 660. In addition, it has its condition code store (CC STORE) hardware connected over lines 2176 through network 670 to the condition code file 660.

The operation of the four-stage pipeline shown in FIG. 21 will now be discussed with respect to the example of Table 1. This operation will be discussed with reference to the information contained in Table 19.

TABLE 19

	Instruction I0, (I1):
Stage 1	Fetch Reg to form Mem-adx
Stage 2	Form Mem-adx
Stage 3	Perform Memory Read
Stage 4	Store RO, (R1)
	Instruction I2:
Stage 1	Fetch Reg RO and R1
Stage 2	No-Op
Stage 3	Perform multiply
Stage 4	Store R2 and CC
	Instruction I3:
Stage 1	Fetch Reg R2 and R3
Stage 2	No-Op
Stage 3	Perform addition
Stage 4	Store R3 and CC
	Instruction I4:
Stage 1	Fetch Reg R4
Stage 2	No-Op
Stage 3	Perform decrement
Stage 4	Store R4 and CC

The operation for each instruction is set forth in Table 19 above.

For instructions I0 and I1, the performance by the processor element 640 in FIG. 21 is the same but for the final stage. The first stage is to fetch the memory address from the register which contains the address in the register file. Hence, stage 1 interconnects circuitry 2140 over lines 2146 through network 670 to that register and downloads it into register 0 from the interface of stage 1. Next, the address is delivered over bus 2144 to stage 2, and the memory write hardware forms the memory address. The memory address is then delivered over bus 2152 to the third stage which reads memory over 2164 through network 650 to the data cache interconnection network 1590. The results of the read operation are then stored and delivered to stage 4 for storage in register R2 which is delivered over lines 2172 through network 672 from register R2 in the register file. The same operation takes place for instruction I1 except that the results are stored in register 1. Hence, the four stages of the pipeline (Fetch, Form Memory Address, Perform Memory Read, and Store The Results) flow through the pipe in the manner discussed. Clearly, when instruction I0 has passed through stage 1, the first stage of instruction I1 commences. This overlapping or pipelining is conventional in the art.

Instruction I2 fetches the information stored in registers R0 and R1 in the register file 660 and delivers them into registers REG0 and REG1 of stage 1. The contents are delivered over bus 2144 through stage 2 as a no operation and then over bus 2152 into stage 3. A multiply occurs with the contents of the two registers, the results are delivered over bus 2162 into stage 4 which then stores the results over lines 2172 through network 670 into register R2 of the register file 660. In addition, the condition code is stored over lines 2176 in the condition code file 660.

Likewise, instruction I3 performs the addition in the same fashion to store the results, in stage 4, in register R3 and to update the condition code for that instruction. Finally, instruction I4 operates in the same fashion except that stage 3 performs a decrement.

Hence, per the example of Table I, the instructions for PEO, as shown in Table 18 are delivered from the PIQO in the following order: I0, I2, and I3. Hence, these instructions are sent through the PEO pipeline stages (S1, S2, S3, and S4) based upon instruction firing times (T16, T17, and T18) as follows:

TABLE 20

PE	Inst	T16					
PE0:	10	S1	S2	S3	S4		
			T17				
	I2		S1	S2	S3	S4	
				T18			
	I3			S1	S2	S3	S4
		T16					
PE1:	I1	S1	S2	S3	S4		
		T16					
PE2:	I4	S1	S2	S3	S4		

Such scheduling is entirely possible since resolution of all data dependencies between instructions and all scheduling of processor resources are performed during TOLL processing prior to program execution. The speed up in processing can be observed in Table 20 since the three firing times (T16, T17, and T18) for the basic block are completed in the cycle time of only six pipeline stages.

The pipeline of FIG. 21 is composed of four equal (temporal) length stages. The first stage 2140 performs the instruction decode and determines what registers to fetch and store as well as performing the two source register fetches required for the execution of the instruction.

The second stage 2150 is used by the computational instructions for the condition code fetch if required. It is the effective address generation stage for the memory interface instructions.

The effective address operations that are supported in the preferred embodiment of the invention are shown below:

1. Absolute address
   The full memory address is contained in the instruction.
2. Register indirect
   The full memory address is contained in a register.
3. Register indexed/based
   The full memory address is formed by combining the designated registers and immediate data.
   a. Rn op K
   b. Rn op Rm
   c. Rn op K op Rm
   d. Rn op Rm op K

In each of the subcases of case 3 above, op may be addition (+), subtraction (−), or multiplication (*). As an example, the addressing constructs presented in the matrix multiply inner loop example are formed from case 3-a where the constant k would be the length of a data element within the array and the operation would be addition (+). These operations are executed in stage two (2150) of the pipeline.

At a conceptual level, the effective addressing portion of a memory access instruction is composed of three basic functions; the designation and procurement of the registers and immediate data involved in the calculation, the combination of these operands in order to form the desired address and the possible updating of any one of the registers involved. This functionality is common in the prior art and is illustrated by the autoincrement and autodecrement modes of addressing available in the DEC processor architecture. See, for example, DEC VAX Architecture Handbook.

Aside from the obvious hardware support required, the effective addressing supported impacts the TOLL software by adding functionality to the memory accessing instructions. In other words, an effective address memory access can be interpreted as a concatenation of two operations, the first the effective address calculation and the second the actual memory access. This functionality can be easily encoded into the TOLL software by one skilled in the art in much the same manner as an add, subtract or multiply instruction would be.

The effective addressing constructs shown are to be interpreted as one possible embodiment of a memory accessing system. Clearly, there are a plethora of other methods and modes for generating a memory address that are known to those skilled in the art. In other words, the effective addressing constructs shown above are shown for design completeness only, and are not to be construed as a key element in the design of the system.

In FIG. 22, are set forth the various structures of data or data fields within the pipeline processor element of FIG. 21. Note that the system is a multiuser system in both time and space. As a result, across the multiple pipelines, instructions from different users may be executing, each with its own processor state. As the processor state is not associated with the processor element, the instruction must carry along the identifiers that specify this state. This processor state is supported by the LRD, register file and condition code file assigned to the user.

Hence, a sufficient amount of information must be associated with each instruction so that each memory access, condition code access or register access can uniquely identify the target of the access. In the case of the registers and condition codes, this additional information constitutes the procedural level (PL) and con-

text identifiers (CI) and is attached to the instruction by the SCSM attachment unit 1650. This is illustrated in FIGS. 22a, 22b and 22c respectively. The context identifier portion is used to determine which register or condition code plane is being accessed. The procedural level is used to determine which procedural level of registers is to be accessed.

Memory accesses require that the LRD that supports the current user be identified so that the appropriate data cache can be accessed. This is accomplished through the context identifier. The data cache access requires that the process identifier (PID) of the current user be available in order to verify that the data present in the cache is indeed the data desired. Thus, an address issued to the data cache takes the form of FIG. 22d. The miscellaneous field is composed of additional information describing the access, e.g., read or write, user or system, etc.

Finally, due to the fact that there are several users executing across the pipelines during a single time interval, information that controls the execution of the instructions that would normally be stored within the pipeline must be associated with each instruction instead. This is reflected in the ISW field shown in FIG. 22a. The information in this field is composed of control fields like error masks, floating point format descriptors, rounding mode descriptors, etc. Each instruction would have this field attached, but, obviously, may not require all the information. This information is used by the ALU stage 2160 of the processor element.

This information as well as the procedural levels, context identification and process identifier are attached dynamically by the SCSM attacher (1650) as the instruction is issued from the instruction cache.

Although the system of the present invention has been specifically set forth in the above disclosure, it is to be understood that modifications and variations can be made thereto which would still fall within the scope and coverage of the following claims.

We claim:

1. A machine implemented method for parallel processing in a plurality of processor elements natural concurrencies in streams of low level instructions contained in programs of a plurality of users, each of the streams having a plurality of single entry-single exit (SESE) basic blocks (BBs), said method comprising the steps of:
   statically adding intelligence representing the natural concurrencies existing within the instructions in each basic block of the programs, said step of adding for each program comprising the steps of:
   (a) ascertaining resource requirements of each instruction within each basic block to determine the natural concurrencies in each basic block,
   (b) identifying logical resource dependencies between instructions,
   (c) assigning condition code storage (CCs) to groups of resource dependent instructions, such that dependent instructions can execute on the same or different processor elements,
   (d) determining the earliest possible instruction firing time (IFT) for each of said instruction in each of said plurality of basic blocks,
   (e) adding said instruction firing times to each instruction in each of said plurality of basic blocks,
   (f) assigning a logical processor number (LPN) to each instruction in each of said basic blocks,
   (g) adding said logical processor numbers to each instruction in each of said basic blocks, and

(h) repeating steps (a) through (g) until all basic blocks are processed for each of said programs, and processing the instructions having the statically added intelligence for executing said programs on a plurality of processor elements (PEs).

2. The method of claim 1 wherein said step of statically adding intelligence further comprises the step of re-ordering said instructions in each of said basic blocks based upon said instruction firing times wherein the instructions having the earliest firing times are listed first.

3. A machine implemented method for parallel processing, in a system, natural concurrencies in streams of low level instructions contained in a program, each of the streams having a plurality of single entry-single exit (SESE) basic blocks (BBs), said system having a plurality of processor elements, said method comprising the steps of:

statically adding intelligence representing the natural concurrencies existing within the instructions in each basic block, said step of adding comprising the steps of:

(a) ascertaining resource requirements of each instruction within each basic block to determine the natural concurrencies in each basic block,

(b) identifying logical resource dependencies between instructions,

(c) assigning condition code storage (CCs) to groups of resource dependent instructions, such that dependent instructions can execute on the same or different processor elements,

(d) determining the earliest possible instruction firing time (IFT) for each of said instructions in each of said plurality of basic blocks,

(e) adding said instruction firing time (IFT) to each instruction in each of said plurality of basic blocks,

(f) assigning a logical processor number (LPN) to each instruction in each of said basic blocks,

(g) adding said logical processor numbers to each instruction in each of said basic blocks, and

(h) repeating steps (a) through (g) until all basic blocks are processed for said program, and

processing the instructions having the statically added intelligence using a plurality of processor elements (PEs).

4. The method according to claim 3 wherein said step of statically adding intelligence further comprises the step of reordering said instructions in each of said basic blocks based upon said instruction firing times wherein the instructions having the earliest firing times are listed first.

5. The method according to claim 3 wherein said step of statically adding intelligence further comprises the step of adding program level information to identify the program level of said instruction.

6. A machine implemented method for parallel processing natural concurrencies in a program using a plurality of processor elements (PEs), said program having a plurality of single entry-single exit (SESE) basic blocks (BBs) with each of said basic blocks (BBs) having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies within said instruction stream in each of said basic blocks (BBs) in said program,

adding intelligence to each instruction in each said basic block in response to the determination of said natural concurrencies, said added intelligence at

least comprising an instruction firing time (IFT) and a logical processor number (LPN) so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor numbers, each of said plurality of processor elements receiving all instructions for that processor in the order of the instruction having earliest instruction firing times, the instruction having the earliest time being delivered first.

7. The method of claim 6 wherein each instruction has a static shared context storage mapping information, and wherein the step of adding intelligence further comprises the step of adding static shared context storage mapping (S-SCSM) information and wherein the step of processing further comprises the step of processing each instruction requiring the shared resources as identified by each said instruction's static shared context storage mapping information, so each program routine can access resources at other procedural levels in addition to the resources at that routine's procedural level.

8. The method of claim 6 wherein the step of determining the natural concurrencies within each basic block comprises the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks,

identifying logical resource dependencies between instructions, and

assigning condition code storage (CCs) to groups of resource dependent instructions.

9. The method of claim 6 wherein the step of adding intelligence to each instruction further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks,

adding said instruction firing times (IFTs) to each instruction in each of said plurality of basic blocks in response to said determination, and

reordering said instructions in each of said basic blocks based upon said instruction firing times.

10. The method of claim 6 wherein the step of adding intelligence to each instruction further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks, and

adding said instruction firing times (IFTs) to each instruction in each of said plurality of basic blocks in response to said determination.

11. The method according to claim 10 further comprising the steps of:

assigning a logical processor number to each instruction in each of said basic blocks, and

adding said assigned logical processor number to each instruction in each of said basic blocks in response to said assignment.

12. The method of claim 6 further comprising the step of forming execution sets (ESs) of basic blocks in response to said step of adding said intelligence wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

13. The method of claim 6 wherein the step of processing further comprises the steps of:

separately storing the instructions with said added intelligence based on the logical processor number, each group of said separately stored instructions containing instructions having only the same logical processor number,

selectively connecting said separately stored instructions to said processor elements, and

each said processor element receiving each instruction assigned to it with the earliest instruction firing time first.

14. The method of claim **13** wherein the step of processing said instruction received by an individual processor element further comprises the steps of:

obtaining the input data for processing said received instruction from shared storage locations identified by said instruction,

storing the results from processing said received instruction in a shared storage identified by said instruction, and

repeating the aforesaid steps for the next instruction until all instructions are processed.

15. A machine implemented method for parallel processing, in a system, natural concurrencies in a plurality of programs of different users with a plurality of processor elements (PEs), each of said programs having a plurality of single entry-single exit (SESE) basic blocks (BBs), with each of said basic blocks (BBs) having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies within said instruction stream in each of said basic blocks in each said program,

adding intelligence to each instruction in each said basic block in response to the determination of said natural concurrencies, said added intelligence representing at least an instruction firing time (IFT) and a logical processor number (LPN) so that all processing resources required by any given instruction are allocated in advance of processing,

processing the instructions having said added intelligence from said programs in said plurality of processor elements corresponding to the logical processor numbers, each of said plurality of processor elements receiving instructions in the order starting with the instruction having earliest instruction firing time, each said processor element being capable of processing instructions from said programs in a predetermined order.

16. The method of claim **15** in which the step of processing further comprises the steps of: dynamically adding context information to each instruction, said dynamically added information identifying the context file in said system assigned to each said program, each of said processor elements being further capable of processing each of its instructions using only the context file identified by said added context information.

17. The method of claim **15** wherein the step of adding intelligence further comprises the adding of program level information to each instruction involved in program level transfers and wherein the step of processing further comprises processing each instruction in communication with a set of shared registers as identified by said instruction's program level information, so that a program routine can access other sets of shared registers in addition to the routine's procedural level set of registers.

18. The method of claim **15** wherein the step of determining the natural concurrencies within each basic block of each program comprises the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks,

identifying logical resource dependencies between instructions, and

assigning condition code storage (CCs) to groups of resource dependent instructions, so that dependent instructions can execute on the same or different processor elements.

19. The method of claim **15** wherein the step of adding intelligence to each instruction in each said program further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks,

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determinination, and

reordering said instructions in each of said basic blocks based upon said instruction firing times.

20. The method of claim **15** wherein the step of adding intelligence to each instruction in each said program further comprises the steps of:

determining the earliest possible instruction firing time for each of said instructions in each of said plurality of basic blocks, and

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determination.

21. The method according to claims **19** or **20** further comprising the steps of:

assigning a logical processor number to each instruction in each of said basic blocks, and

adding said assigned logical processor number to each instruction in each of said basic blocks in response to said assignment.

22. The method of claim **15** further comprising the step of forming execution sets (ESs) of the basic blocks for each said program in response to said step of adding said intelligence wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

23. The method of claim **15** wherein the step of processing further comprises the steps of:

separately storing the instructions with said added intelligence in a plurality of sets and wherein said separate storage of each set is based on the assigned logical processor number,

selectively connecting said separately stored instructions in said sets to said processor elements based on said logical processor number, and

each said processor element receiving the instruction having the earliest instruction firing time first.

24. The method of claim **23** wherein the step of receiving said instruction from said separately stored instructions in said sets by an individual processor element further comprises the steps of:

obtaining input data for processing said received instruction from shared storage locations identified by said instruction,

storing the results based from processing said received instruction in a shared storage identified by said received instruction, and

repeating the aforesaid steps for the next received instruction from said next set based upon said predetermined order until all instructions are processed in said sets.

25. A method for parallel processing natural concurrencies in a plurality of programs with a plurality of processor elements (PEs), said processor elements having access to input data located in a plurality of shared resource locations, each of said programs having a plurality of single entry-single exit (SESE) basic blocks (BBs) with each of said basic blocks (BBs) having a stream of instructions, said method comprising the steps of:

ascertaining the resource requirements of each instruction within each of said basic blocks for each said program,

determining, based on said resource requirements, the earliest possible instruction firing time (IFT) for each of the instructions in each of said plurality of basic blocks,

adding said instruction firing times to each instruction in each of said plurality of basic blocks in response to said determination,

assigning a logical processor number (LPN) to each instruction in each of said basic blocks,

adding said assigned logical processor number to each instruction in each of said plurality of basic blocks in response to said assignment,

separately storing the instructions with said added instruction firing time and said added logical processor numbers in a plurality of sets, each set having a plurality of separate storage regions, wherein at least one set contains at least one of said programs and wherein said separate storage in each set is based on the logical processor number,

selectively connecting said separately stored instructions for one of said sets to said processor elements based on the logical processor number, and

each said processor element receiving each instruction to be connected thereto with the earliest instruction firing time (IFT) first, said processor element being capable of performing the steps of:

(a) obtaining said input data for processing said received instruction from a shared storage location in said plurality of shared resource locations identified by said instruction,

(b) storing the results based from processing said received instruction in a shared storage location in said plurality of shared resource locations identified by said received instruction, and

(c) repeating the aforesaid steps (a) and (b) for the next received instruction from one of said sets based upon a predetermined order until all instructions are processed.

26. The method of claim 25 further comprising the steps of forming execution sets (ESs) of basic blocks in response to said steps of adding said instruction firing times and logical processor numbers wherein branches from any given basic block within a given execution set to a basic block in another execution set is statistically minimized.

27. The method of claim 25 wherein the step of adding intelligence further comprises the step of adding static shared context storage mapping information and wherein the step of processing further comprises the step of processing each instruction requiring at least one shared storage location, said at least one location corresponding to said instruction's static shred context state mapping information, so each program routine can access its set of storage locations and, in addition, at

least one set of shared storage locations at another procedural level.

28. A system for parallel processing natural concurrencies in a program, said program having a plurality of single entry-single exit (SESE) basic blocks (BBs) wherein each of said basic blocks (BBs) contains a stream of instructions, said system comprising:

means receptive of said plurality of basic blocks for determining said natural concurrencies within said instruction stream for each of said basic blocks, said determining means further adding at least an instruction firing time (IFT) and a logical processor number (LPN) to each instruction in response to said determined natural concurrencies so that all processing resources required by any given instruction are allocated in advance of processing,

means receptive of said basic blocks having said added instruction firing times and logical processor numbers for separately storing said received instructions based upon said logical processor number, and

a plurality of processor elements (PEs), connected to said storing means based upon said logical processor numbers, each of said processor elements processing its received instructions and the order of processing those instructions being that the instructions with the earliest instruction firing time are processed first.

29. The parallel processor system of claim 28 in which:

said determining means further has means for adding program level information (S-SCSM) to said instructions, said information containing level information for each said instruction in order to identify the relative program levels required by the instruction within said program,

a plurality of sets of registers, said registers having a different set of registers associated with each said program level, and

said processor elements being further capable of processing each of its received instructions in at least one set of registers identified by said received instruction.

30. The system of claim 28 wherein said determining means further has means for forming the basic blocks containing said added instruction firing times and logical processor numbers into execution sets (ESs), wherein branches from any given basic block within a given execution set out of said given execution set to a basic block in another execution set is statistically minimized.

31. The system of claim 30 wherein said determining means is further capable of attaching header information to each formed execution set, said header at least comprising:

(a) the address of the start of said instructions, and

(b) the length of the execution set.

32. The system of claim 30 further comprising storing means having:

a plurality of caches receptive of said execution sets for storing said instructions,

means connected to said caches for delivering said instructions stored in each of said caches to said plurality of processor elements (PEs), and

means connected to said caches and said delivering means for controlling said storing and said delivery of instructions, said controlling means being further capable of executing the branches from individual basic blocks.

33. The system of claim 28 wherein each of said plurality of processor elements is context free in processing a given instruction.

34. The system of claim 28 wherein said plurality of shared resources comprises:

a plurality of register files, and

a plurality of condition code files, said plurality of register files and said plurality of condition code files together with said storing means being capable of storing all necessary context information for any given instruction during the processing of said instruction.

35. A machine implemented method for parallel processing streams of low level instructions contained in the programs of a plurality of users for execution in a system having a plurality of processor elements and shared storage locations, each of the streams having a plurality of single entry-single exit basic blocks, said method comprising the steps of:

statically adding intelligence to the instructions in each basic block of the programs, said added intelligence being responsive to natural concurrencies within said instruction streams, said added intelligence representing at least an instruction firing time for each said instruction, and

processing the instructions having the statically added intelligence for executing the programs, the step of processing further comprising the steps of:

(a) delivering the instructions to the system,

(b) assigning each said user program to a context file in said system, whereby each program during execution has its own identifiable context file,

(c) dynamically generating shared context storage mapping information for said instructions identifying said context file being assigned to said instructions and containing shared storage locations,

(d) separately storing the delivered instructions in the system based on a processor on which the instruction is to be executed,

(e) selectively connecting the separately stored instructions to the assigned processor for the instructions, said separately stored instructions being delivered in order of earliest instruction firing times of the separately stored instructions,

(f) sequentially processing said instructions from each connected separately stored instructions in each said connected processor element,

(g) obtaining any input data for processing said connected instruction from a shared storage location identified, at least in part, by said shared context storage mapping information,

(h) storing the results of processing a said connected instruction in a shared storage location identified at least in part by said shared context storage mapping information, and

(i) repeating steps (a) through (h) until all instructions of each of said plurality of basic blocks for all of said programs are processed.

36. A machine implemented method for parallel processing a stream of instructions having natural concurrencies in a parallel processing system having a plurality of processor elements, said stream of instructions having a plurality of single entry-single exit basic blocks, said method comprising the steps of:

determining natural concurrencies within said instruction stream,

adding intelligence to each instruction stream in response to the determinination of said natural concurrencies, said added intelligence comprising at least an instruction firing time and a logical processor number for each instruction so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor number, each of said plurality of processor elements receiving all instructions for that processor element in order according to the instruction firing times, the instruction having the earliest time being delivered first.

37. A machine implemented method for parallel processing a plurality of programs of different users in a system having a plurality of processor elements, each of said programs having a plurality of single entry-single exit basic blocks, with each of said basic blocks having a stream of instructions, said method comprising the steps of:

determining the natural concurrencies among the instructions in each said program,

adding intelligence to said basic blocks in response to the determination of said natural concurrencies, said added intelligence comprising at least an instruction firing time and a logical processor number so that all processing resources required by any given instruction are allocated in advance of processing, and

processing the instructions having said added intelligence in said plurality of processor elements corresponding to the logical processor number, each of said plurality of processor elements receiving instructions in accordance with the instruction firing times, starting with the instruction having earliest instruction firing time.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO.	: 5,021,945	Page 1 of 31
DATED	: June 4, 1991	
INVENTOR(S)	: Gordon E, Morrison	

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

The title page should be delete to appear as per attached title page.

Column 1-60 should be deleted to appear as per attached columns 1-56.

Signed and Sealed this

Tenth Day of November, 1992

Attest:

DOUGLAS B. COMER

Attesting Officer

Acting Commissioner of Patents and Trademarks

BIA0001172

United States Patent [19]

Morrison et al.

[11] **Patent Number: 5,021,945**

[45] **Date of Patent: Jun. 4, 1991**

[54] **PARALLEL PROCESSOR SYSTEM FOR PROCESSING NATURAL CONCURRENCIES AND METHOD THEREFOR**

[75] Inventors: **Gordon E. Morrison**, Denver; **Christopher B. Brooks; Frederick G. Gluck**, both of Boulder, all of Colo.

[73] Assignee: **MCC Development, Ltd.**, Boulder, Colo.

[21] Appl. No.: **372,247**

[22] Filed: **Jun. 26, 1989**

Related U.S. Application Data

[62] Division of Ser. No. 794,221, Oct. 31, 1985, Pat. No. 4,847,755.

[51] **Int. Cl.**5 .... **G06F 15/16**; G06F 9/38

[52] **U.S. Cl.** **364/200**; 364/230.3; 364/228.2; 364/262.4; 364/271.3

[58] **Field of Search** ... 364/200 MS File, 900 MS File

[56] **References Cited**

U.S. PATENT DOCUMENTS

3,343,135	9/1967	Freiman et al.	364/200
3,611,306	10/1971	Reigel .	
3,771,141	11/1973	Culler .	
4,104,720	8/1978	Gruner .	
4,109,311	8/1978	Blum et al. .	
4,153,932	5/1979	Dennis et al. .	
4,181,936	1/1980	Kober .	
4,228,495	10/1980	Bernhard .	
4,229,790	10/1980	Gilliland et al. .	
4,241,398	12/1980	Caril .	
4,270,167	5/1981	Koehler et al. .	
4,430,707	2/1984	Kim .	
4,435,758	3/1984	Lorie et al. .	
4,466,061	8/1984	De Santis .	
4,468,736	8/1984	De Santis .	
4,514,807	4/1985	Nogi .	
4,574,348	3/1986	Scallon .	

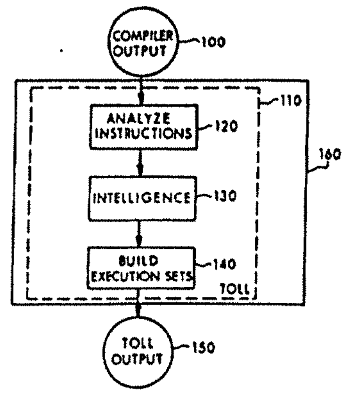
OTHER PUBLICATIONS

Dennis, "Data Flow Supercomputers", Computer, Nov. 1980, pp. 48–56.
Hagiwara, H. et al.; "A Dynamically Microprogrammable, Local Host Computer With Low-Level Parallelism", IEEE Transactions on Computers, C-29, N-7, Jul. 1980, pp. 577–594.
Fisher et al., "Microcode Compaction: Looking Backward and Looking Forward", National Computer Conference, 1981, pp. 95–102.
Fisher et al., "Using an Oracle to Measure Potential Parallelism in Single Instruction Stream Programs", IEEE No. 0194–1895/81/0000/0171, 14th Annual Microprogramming Workshop, Sigmicro, Oct. 1981, pp. 171–182.
J. R. Vanaken et al., "The Expression Processor", IEEE Transactions on Computers, C-30, No. 8, Aug. 1981, pp. 525–536.
Bernhard, "Computing at the Speed Limit", IEEE Spectrum, Jul. 1982, pp. 26–31.
Davis, "Computer Architecture", IEEE Spectrum, Nov. 1983, pp. 94–99.
Hagiwara, H. et al., "A User-Microprogrammable Local Host Computer With Low-Level Parallelism", Article, Association for Computing Machinery, #0149–7111/83/0000/0151, 1983, pp. 151–157.
Mc Dowell, Charles Edward, "SIMAC: A Multiple ALU Computer", Dissertation Thesis, University of California, San Diego, 1983 (111 pages).
Mc Dowell, Charles E., "A Simple Architecture for Low Level Parallelism", Proceedings of 1983 International Conference on Parallel Processing, pp. 472–477.
Requa et al., "The Piecewise Data Flow Architecture: Architectural Concepts", IEEE Transactions on Computers, vol. C-32, No. 5, May 1983, pp. 425–438.
Fisher, A. T., "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", Computer, 1984, pp. 45–52.
Fisher et al., "Measuring the Parallelism Available for Very Long Instruction, Word Achitectures", IEEE Transactions on Computers, vol. C-33, No. 11, Nov. 1984, pp. 968–976.

Primary Examiner—Eddie P. Chan
Attorney, Agent, or Firm—Hale and Dorr

[57] **ABSTRACT**

A computer processing system containing a plurality of identical processor elements each of which does not retain execution state information from prior operations. The plurality of identical processor elements op-



erate on a statically compiled program which, based upon detected natural concurrencies in the basic blocks of the programs, provide logical processor numbers and an instruction firing time to each instruction in each basic block. Each processor element is capable of executing instructions on a per instruction basis such that dependent instructions can execute on the same or different processor elements. A processor element is capable of executing an instruction from one context followed by an instruction from another context through use of shared storage resources.

37 Claims, 17 Drawing Sheets